jamk.fi

Creating network services with actor model

Iiro Surakka

Bachelor's thesis
May 2016
Technology, communication and transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Surakka, Iiro	Bachelor's thesis	24.5.2016
		Language of publication: English
	Number of pages: 31	Permission for web publication: x

Title of publication

Creating network services with actor model

Degree programme

Degree Programme in Software Engineering

Supervisor(s)

Salmikangas, Esa

Assigned by

Nodeon Finland Oy

Abstract

Networked services can be built with variety of frameworks and libraries. One of these frameworks is the actor model. The work was assigned by Nodeon Finland Oy which is an engineering and software development company which specializes in industrial internet and smart traffic solutions.

The objective was to use and explore the actor model in smart traffic solutions and other network services. The purpose was to create guidelines for the company to provide new and existing employees with material when developing network services. Also, the goal was to improve the quality of software and its development in the field of smart traffic by giving concrete examples of how the actor model can be used.

Various software solutions were built utilizing the actor model in order to gain experience with its use and to develop useful patterns. These solutions were analyzed to determine whether the actor model was an overall improvement.

As result the company gets information about the actor model and what kind of software can be built with it. The guidelines can also be used when maintaining already developed solutions to get insight into the reasons behind various architectural decisions. The result can be used to determine whether the actor model is worth using in the future and whether the advantages outweigh the disadvantages.

Keywords/tags (subjectshttp://vesa.lib.helsinki.fi/)

C#, software development, software architecture, software

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Surakka, Iiro	Opinnäytetyö, AMK	24.5.2016
	Sivumäärä 31	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Creating network services with actor model

Tutkinto-ohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Esa Salmikangas

Toimeksiantaja(t)
Nodeon Finland Oy

Tiivistelmä

Verkkoon yhteydessä olevia sovelluksia voidaan kehittää monilla eri ohjelmistokehyksillä ja kirjastoilla. Yksi näistä ohjelmistokehyksistä on toimijamalli. Työn toimeksiantaja on Nodeon Finland Oy, joka on teolliseen internetiin ja älyliikenteeseen erikoistuva insinööri ja sovelluskehitys yritys.

Tavoitteena oli käyttää ja tutkia toimijamallia älyliikenteen ratkaisuissa ja muissa verkkosovelluksissa. Tämän lisäksi tarkoitus oli luoda ohjeet yritykselle joita voidaan käyttää uusien ja vanhojen työntekijöiden perehdyttämisessä verkkosovelluksia kehittäessä. Älyliikenteen sekä muiden alojen sovellusten laatua ja niiden kehitystä oli myös tarkoitus kehittää antamalla konkreettisia esimerkkejä kuinka toimijamallia voidaan käyttää.

Tulosten saavuttamiseksi erilaisia toimijamallia käyttäviä sovellusratkaisuja toteutettiin kokemuksen kartuttamiseksi ja sunnittelumallien kehittämiseksi. Näitä ratkaisuja analysoitiin, jotta voitiin päättää oliko toimijamallilla kehitetty sovellus kokonaisuudessaan parannus.

Tuloksena yritys saa tietoa toimijamallista ja siitä minkälaisia sovelluksia sillä voi tehdä. Ohjeita voidaan myös käyttää kun ylläpidetään toimijamallilla kehitettyjä sovelluksia sekä saadaan tietoa syihin sovelluksissa tehtyihin arkkitehtuurisiin ratkaisuihin. Tulosten avulla voidaan myös päättää kannattaako toimijamallia käyttää tulevaisuudessa ja ovatko sen tuomat hyödyt haittojen arvoisia.

Avainsanat (asiasanat)
C#, ohjelmistokehitys, sovelluskehykset, verkko-ohjelmointi

Muut tiedot

Contents

Figures

Tables

Terminology

API

API (Application User Interface) is an interactable interface exposed by an application. Consumers of API can interact with the application through the API.

GUI

Graphical User Interface is a visual method of displaying text and images to the user. GUI usually involves interaction between the user and the computer. Many user-friendly software has some kind of GUI.

HTTP

HTTP (Hypertext Transfer Protocol) is the main protocol used in World Wide Web. It is text based protocol for communicating between webservers and their users.

LINQ

Language-Integrated Query is a feature in C# which makes querying various data-structures more pleasing by introducing various extension methods for filtering and providing a syntax similar to ones used with relational databases query languages.

REST

REST (Representational State Transfer) is software architectural style used in HTTP based services. It standardizes how HTTP services should define their interface.

SCADA

Supervisory Control And Data Acquisition is a system designed for controlling and monitoring of industrial devices.

Windows

Widely used operating system developed by Microsoft. Used mostly in desktop environments but servers are also popular.

WPF

Windows Presentation Foundation is a graphical user interface library for Windows desktop applications. Layouts are defined in XAML which is XML-based language. Interactions are programmed in .NET based languages.

1 Introduction

Organizations use variety of languages, libraries and frameworks to develop their applications. All have their own advantages and disadvantages. This thesis takes a look at how network services can be developed using actor model as the underlying framework.

The work was assigned by Nodeon Finland Oy which is an engineering and software development company which specializes in industrial internet and smart traffic solutions. The subject was chosen as great deal of software development had happened utilizing the actor model.

The goal was to use the actor model in practice, improve its usage and analyze what kind of patterns and limitations it has. The goal was to create guidelines for developing software using the actor model and to improve quality of smart traffic solutions. These guidelines would allow the quality of software to be improved in the field on smart traffic.

2 Technologies

In development of network services, various technologies were used to develop efficient, reliable and error tolerant software. While some of them are necessary to build software in the first place, some were chosen to ensure these requirements were met.

2.1 C#

C# is a general purpose programming language developed by Microsoft. It encompasses multiple programming paradigms such as strong typing, imperative, declarative, functional, generic and object-oriented disciplines. The language is designed to be robust and programming productivity is important. (Wikipedia, C# 2016)

C# syntax is expressive; however, it is also easy to learn. It is similar to languages like C, Java and C++. It simplifies many C++ concepts and comes with many features which Java lack. (MSDN 2016)

The main programming paradigm of the language is object-oriented. It supports all the features which object-oriented languages should have such as encapsulation, inheritance, and polymorphism. Classes may only inherit from one class, unlike C++. To complement this, any number of interfaces can be implemented by classes. Generics are also supported which allows classes to be parameterized with types for more compile time safety. (MSDN 2016)

C# is widely used in all kinds of programming projects by many organizations. New versions with more features are being constantly developed. Currently planned, upcoming features include pattern matching, first-class tuples and local functions (Channel9, The Future of C# 2016).

2.2 .NET Framework

.NET Framework is a standard library designed to be used with .NET based languages such as C#, Visual Basic and F#. It contains common functionality needed in most software and also advanced domain specific features. (Figure 1)

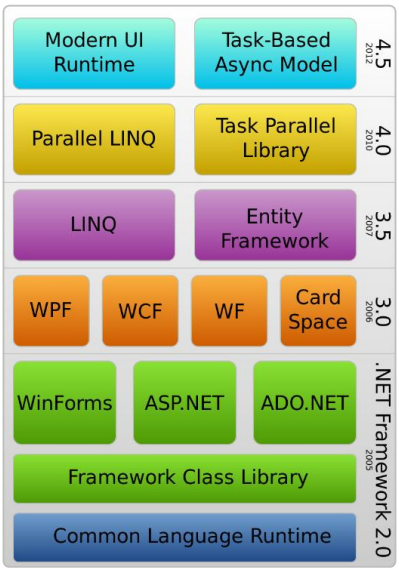

Figure 1 .NET Framework components

(Soumyasch 2007)

Microsoft has been pushing .NET forward by open sourcing components that have historically always been closed source. Roslyn, a new compiler with many improvements has been developed and made open-source by Microsoft.

Roslyn provides a variety of interfaces to internals of the compiler for use with development tools. These tools can be used to analyze and modify codebases of developers. Many of the features of Visual Studio have been rewritten to take advantage of Roslyn. (Roslyn 2016)

2.3 Visual Studio

Visual Studio is an integrated development environment mainly for .NET based languages. Additionally, languages such as C, C++, Python, Ruby, Node.js, JavaScript and CSS are supported. (Wikipedia, Microsoft Visual Studio) In addition to text editing, the IDE has many useful features for software development such as powerful debugger, unit testing, profiling, GUI designer, project management and version control support. The editor is also highly customizable and an example of this can be seen in figure 2.

One of the most useful features in Visual Studio is IntelliSense. IntelliSense can suggest method, property, field etc. names as they are typed partially and with a press of a button the name is autocompleted.

Visual Studio is highly extendable and various plugins have been developed to make the development experience more pleasant. Microsoft has a Visual Studio Gallery which is the central location for finding information about extensions (Wikipedia, Microsoft Visual Studio).

An example of a great extension for Visual Studio is Resharper. It is one of the most popular extensions for Visual Studio. It extends the refactoring and analyzing features of the IDE. Resharper is developed by JetBrains which has released IDEs for multiple programming languages.

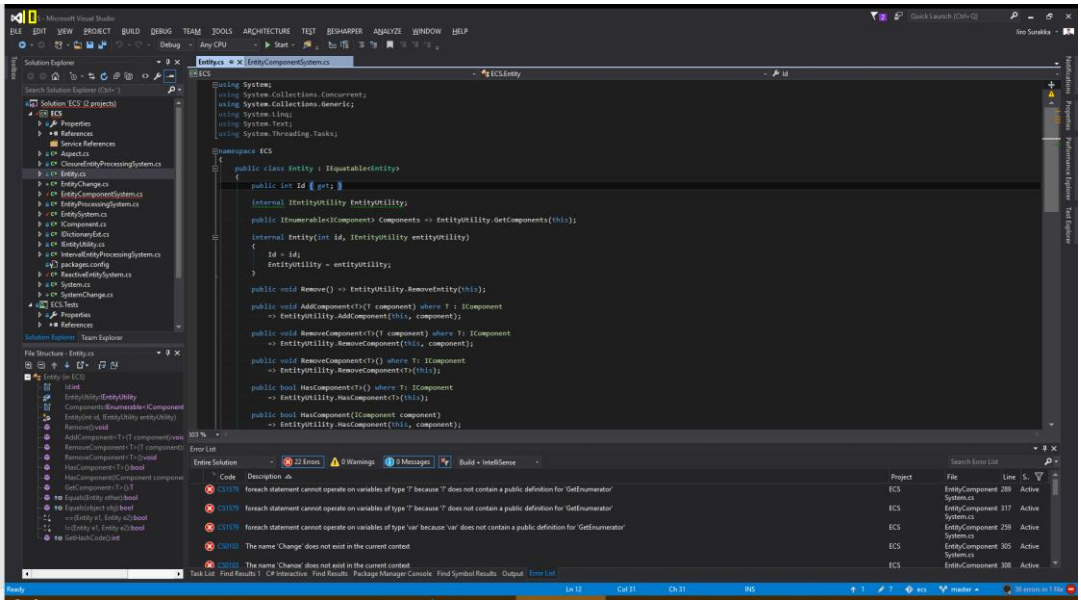

Figure 2 Visual Studio 2015 with the dark theme

2.4 NuGet

Most modern languages have some kind of package management system for dependencies. NuGet is a package manager for .NET based projects. For .NET based languages it was first introduced in 2010 (Xavier Decoster 2016).

Dependencies can be installed with a click of a button. Packages are automatically downloaded and built in projects which use NuGet. All of the libraries used in this thesis can be found in NuGet.

2.5 Topshelf

Topshelf is an open source library for wrapping applications to Windows services. Normally setting up an application to be service involves great deal of work. Topshelf makes managing Windows service easy, removing all the cruft involved.

After the installation of Topshelf, console applications can be run and debugged normally and be installed with a single command as a service.

2.6 NLog

NLog is a widely supported logging library for .NET based languages. It is highly configurable. Formats, logging targets and even logging interfaces can be customized. It also has integrations with many frameworks including Akka.NET.

3 Actor model

Actor model is a high level abstraction for writing concurrent and distributed systems. It makes development of concurrent and parallel systems easier by alleviating the developer from having to deal with synchronization primitives. (Akka.NET 2016)

Actor model simplifies thinking when making concurrent applications by enforcing certain program structure from the developer. Building software with this ideology lets the underlying actor model implementation optimize performance in a certain way because it knows the developer has built it within the constraints of actor model.

3.1 Actors

Actors are the most basic unit of actor model. They have a state and modify it based on received messages. The state is never modified outside of message receiving. This property allows the actor model implementation to process messages going to different actors concurrently.

Actors form a hierarchy. Each actor can have any amount of child actors. Each actor has a single parent. (Figure 3) Creating another actor inside an actor implicitly makes it a child of the creating actor.

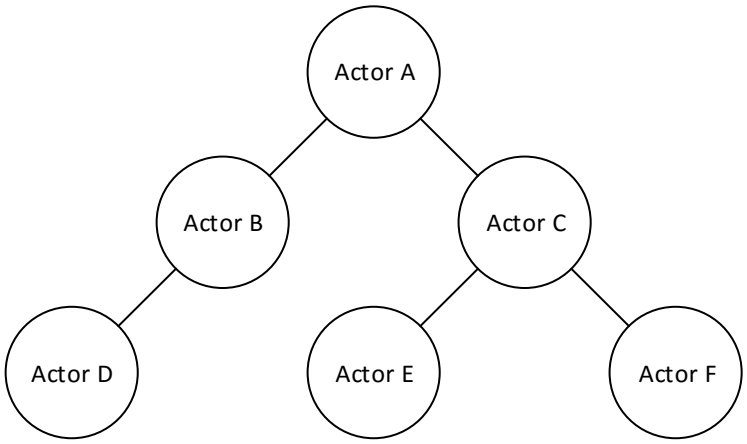

Figure 3 Actor hierarchy

The parent of an actor supervises the work of children. Instead of doing plenty of work in parent it is useful to delegate work to child actors. When a failure happens inside an actor it only affects the single actor. The parent decides what to do when such a failure happens.

Each actor has an address. If an actor knows the address of any other actor, they can send messages to it. Messages always come with sender address. The receiving actor can send back messages to the sender address.

3.2 Messages

Messages are the method of communication between actors. Messages are sent by actors to other actors. All processing takes place in reaction to receiving a message.

Message sending is one-way and it occurs asynchronously. Sending a message has no required overhead in the actor model like a use of synchronization primitives. Implementations are free to use these primitives for optimization purposes. (Carl Hewitt 2010)

The actor model messages have the following axioms which define what an actor can do when a message is received:

- create more actors
- send messages to actors is has addresses to
- designate what to do with the next message

(Channel9, The Actor Model 2016)

Sending a message takes place in a fire and forget manner. Messages are guaranteed to be received in at-most-once manner (Channel9 2016). This means a message may be lost or reach its destination; however, it may not duplicate itself at the receiving end. There is no guarantee that a message will ever reach its destination. However certain actor model implementations may provide such guarantees.

Messages can be compared with function calls. Sending a message causes something to occur in an actor, just like a function call. They can also have return values although they are usually abstraction built on top of an actor model. The advantage with messages is that they are always thread-safe and they can be sent to another process or even over network.

Messages need to be immutable because they can be sent anywhere. (Figure 4) This is so that the receiver does not accidentally modify the message and cause unsafe side effects. This lets actor model implementations pass around the same instance of message around as an optimization without problems.

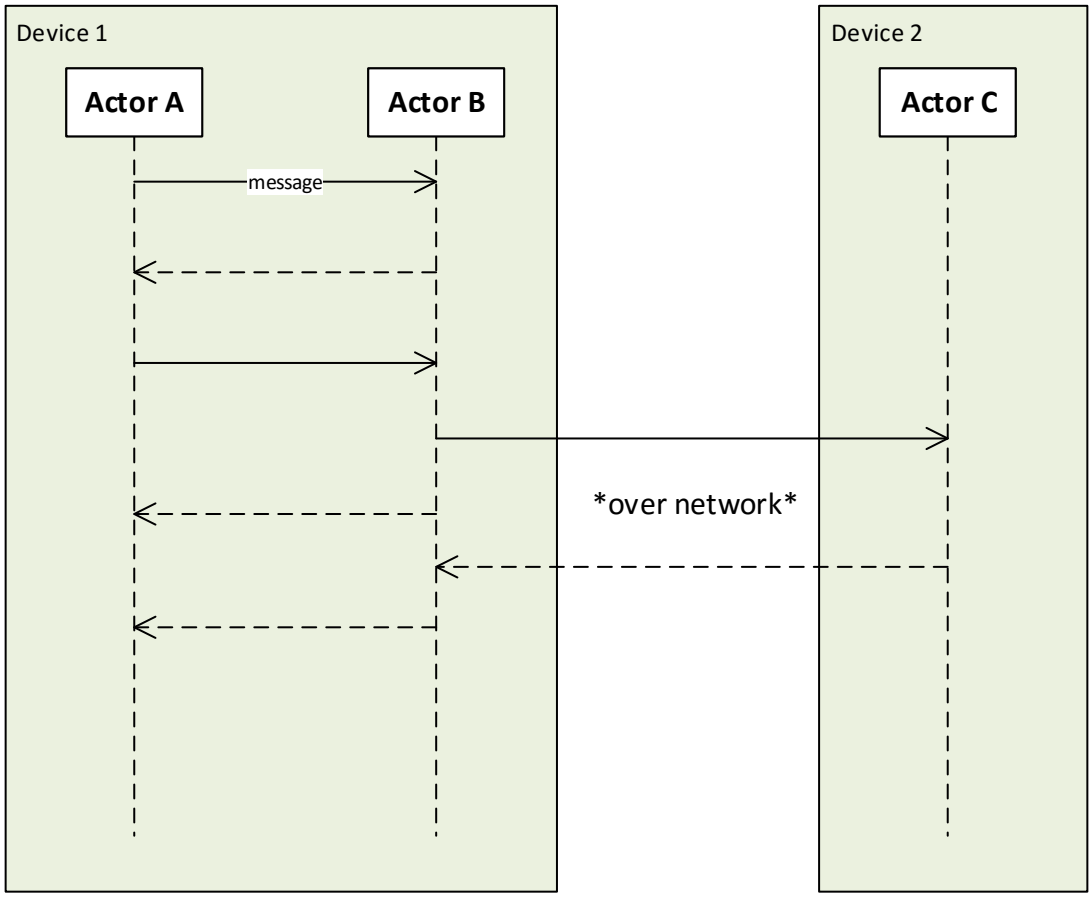

Figure 4 Sending messages to other actors

3.3 Multithreading

Actor model is inherently easy to multithread and create distributed systems with. Actors process messages one by one so the developer does not need to be concerned with concurrency problems in the context of a single actor. In turn, message processing of each actor can be run in parallel and synchronization can take place inside the actor model implementation. This allows developers to build high throughput applications easily.

3.4 Testing

Actors are self-contained pieces of functionality where the defined types of messages to receive define the interactable interface. It is very easy to swap actor out as long as the same message types are handled, which makes testing easy as mock actors can be created for testing and they can be swapped with real actors. Messages and actions taken upon them can be tested by sending messages and observing the responses.

If an application connects to a service but the service cannot be run for some reason a good candidate would be to make a mock actor for testing. The service could be sending connected, disconnected, state messages. The mock actor could send the exact same messages with generated values. This way the service consumer can be tested even if the real service cannot be run.

3.5 Limitations

While the actor model is great for distributed and concurrent applications it poses many limitations to the developer. These limitations must be taken into account when deciding whether to use actor model in your application.

Doing concurrent programming outside of actors can be problematic. While actors can process messages concurrently the processing itself should run sequentially and relatively fast. Actor model implementations carefully schedule messages for maximum efficiency. Blocking for a long time can cause problems with scheduling so it is

recommended to run concurrent tasks in the background and schedule a message on completion.

Another problem is efficiency. Messages can be compared to function calls. Message, in a sense, is a function parameter and there can be a response to a message which acts as a return value. If a program is relatively small and does not need to do great deal of concurrent work, actors are much slower compared to normal function calls. Also, a carefully constructed multithreaded program will always be faster than a program which uses the actor model. The maintenance cost will be higher in a handcrafted multithreaded program but it will be faster.

The actor model is a simple concept to understand and utilize. Working with the actor model is really easy. If you need to build concurrent and distributed software the actor model is a great candidate. However, I think it needs to be carefully considered whether to use the actor model because if the application doesn't need to be highly concurrent or distributed it is probably not worth it. Most people have better understanding of normal programming than the actor model so they can work faster without it. Also, migrating away from the actor model is much harder than the other way around.

3.6 Akka.NET

Akka.NET is open source .NET implementation of an actor model. It is a port of the Java/Scala version of Akka. Akka and Akka.NET are the most popular actor model libraries in their respective programming languages.

A distinctive difference in Akka.NET when compared to traditional actor model is the concept of supervision, fault tolerance and location transparency. These concepts are not normally described when discussing the actor model.

The main use cases for Akka.NET are investment and merchant banking, social media, gaming, automobile and traffic systems, health care and data analytics. Also, any system which requires slow latency and high throughput are good candidates for Akka.NET. (Akka.NET, Why Akka 2016)

Akka.NET can be easily included in existing projects. The only needed parts of Akka.NET can be used. Akka.NET provides good performance even if it is run on one machine only. (Akka.NET, Why Akka 2016)

3.6.1 Fault tolerance

Actors have a method of tolerating errors. The parent of an actor can define what to do when a child throws an error. Exceptions are not handled with the traditional try-catch mechanism used in normal C# programs. Possible actions on error are resuming, stopping or restarting the actor. The selected action also propagates to all children.

Resuming just discards the message that caused the error. Following messages are processed normally. Usually this is used when the error is minor and does not cause the actor's state to be corrupted.

Stopping the actor prevents all future messages to be processed by the offending actor. Usually this is used if there is some kind of custom logic to restart the actor.

Restarting the actor causes it to be recreated with the same constructor parameters as it was originally created, which is usually the most common way of handling errors as recreation should always be safe.

A good example of fault tolerance would be an actor which supervises some kind of connection (Figure 5). The connection could throw all kinds of exceptions and the parent decides what to do in each case.

```
protected override SupervisorStrategy SupervisorStrategy()
{
    return new OneForOneStrategy(
        maxNrOfRetries: -1,
        withinTimeRange: null,
        localOnlyDecider: ex =>
        {
            // These exceptions are not fatal
            if (ex is ApplicationException) return Directive.Resume;

            // Connection is probably dead
            if (ex is SocketException) return Directive.Restart;

            // Otherwise let parent decide
            return Directive.Escalate;
        });
}
```

Figure 5 Deciding what to do when an exception occurs

3.6.2 Location transparency

One advantage of using the actor model is the ability for actors to reside in another thread, process or even physical location. Messages can be sent to any actor as long as their address is known, which makes scaling relatively easy as there is no need to change the program in order to scale for multiple devices.

The sending actor does not need to know whether the receiving address is on the same machine or different one. Serialization and deserialization of messages occurs automatically and transparently for both sender and receiver.

While the actor model messages must be immutable and serializable it is not enforced by Akka.NET until a message is actually sent over network or to another process. Akka.NET has a setting to enforce serializability in every situation but it should only be used for testing purposes as messages may be passed without serialization between actors if the messages lie in same program for optimization purposes. (Akka.NET, Serialization 2016)

The used serialization library in Akka.NET is configurable. Akka.NET includes binary and JSON based serializers. Custom serializers are also easy to build. The default serializer is JSON based using *Newtonsoft.Json* although that may change in future. (Akka.NET, Serialization 2016)

3.6.3 Akka.IO

Akka.NET contains networking capabilities abstracted under Akka.NET actor model. This abstraction is called Akka.IO. Akka.IO allows sending of network packets using normal actor model messages.

In Akka.IO, once a TCP connection is established an actor is formed which represents the connection. Sending *Tcp.Write* messages to this actor causes data to be sent to the actual TCP connection. Similarly, the connection actor can send back *Tcp.Received* messages which contain the data sent from connection.

3.6.4 Integrating with different libraries

Normal libraries can be wrapped to actors in order to take full advantage of actor model. Doing this is useful as otherwise using the libraries could be awkward.

WPF

The main concern with WPF integration is running GUI interactions in main thread. Akka.NET actors cannot update GUI elements so they must schedule the updates on the main thread somehow. Reactive extensions library is a good candidate for doing these updates.

Reactive extensions

Reactive extensions provide a functional event system implementation. Compared to reactive extensions, .NET version of events are hard and awkward to use. Reactive extensions observers and observables can be passed around and they have LINQ-like methods for filtering. These properties make reactive extensions usable with Akka.NET.

Actors can receive *IObservable* objects in order to subscribe them for events which happen inside the actor (Figure 6). It is important that the sender of IObservable subscribes to it on the main thread if it is used with WPF.

```
var numberSubject = new Subject<int>();

Receive<IObservable<int>>(observable =>
{
    observable.Subscribe(numberSubject);
});
```

Figure 6 Subscribing message sender to events

Web API 2

Web API 2 integrates relatively well with the actor model as Web API 2 fully supports asynchronous tasks. Akka.NET actors can respond to messages if the message was sent using the “Ask” method. This way Web API routes can use response messages from actors in HTTP responses.

When using the actor model with Web API 2 it is usually not beneficial to put meaningful data to HTTP responses because the request processing may take varying

amounts of time. Instead, some kind of notification system could be more useful where the HTTP server does HTTP request to send the result.

3.6.5 Example

Simple example of Akka.NET demonstrating basic features can be seen in figure 7. The example has actor classes *Pinger* and *Ponger.* On construction the Pinger creates a Ponger as a child actor. The Pinger also receives *Ping* and *Pong* messages. Once a Ping message is received, it is simply tells the same message to the Ponger child. If the Pinger receives a Pong message it increments its state by one. The Ponger receives Ping messages and tells back Pong messages in exchange.

In the main function, an actor system is created, one Pinger actor is created which in turn creates the Ponger child for it and then a ping message is sent to the created actor to start the system. The program flow then goes as follows:

1. Pinger gets the ping message from main method and sends it to the child Ponger
2. The child Ponger gets the Ping message and sends back a Pong message
3. The Pinger receives the Pong message and increments the state by one

```
class Program
{
    static void Main(string[] args)
    {
        // Create actor system
        var actorSystem = ActorSystem.Create("ping-test");

        // Create a Pinger actor
        var pinger = actorSystem.ActorOf<Pinger>();

        // Send Ping message to the pinger actor
        pinger.Tell(new Ping());

        Console.ReadKey();
    }
}

class Pinger : ReceiveActor
{
    private int state;
    private IActorRef ponger;

    public Pinger()
    {
        // Create child actor ponger
        ponger = Context.ActorOf<Ponger>();

        // Upon receiving a ping message, send the message to ponger
        Receive<Ping>(m =>
        {
            ponger.Tell(m);
        });

        // Upon receiving a pong message, increment state by 1
        Receive<Pong>(m =>
        {
            state += 1;
        });
    }
}

class Ponger : ReceiveActor
{
    public Ponger()
    {
        // Upon reciving a ping message, send pong message back to the sender
        Receive<Ping>(m =>
        {
            Sender.Tell(new Pong());
        });
    }
}

// The messages
class Ping {  }
class Pong {  }
```

Figure 7 An example demonstrating Akka.NET

4 Networking

Networking with Akka.NET and Akka.IO is easy as long as actor model is understood and there is a basic understanding of the network protocols involved. This section focuses mostly on networking with TCP protocol.

4.1 Handling connection state changes

Establishing a TCP connection with Akka.IO is easy. A message is sent to connect and a connected message is waited for. There is not much that can go wrong. If someone is already using the connection it might be useful, depending on the desired outcome, to buffer their messages until the connection is established. An advantage with Akka.NET is that messages are automatically buffered until an actor is initialized.

TCP disconnection is both symmetric and independent. Symmetric as in either side of the connection can initiate the disconnect and independent as in one side can close the connection without the other side realizing this. The closing side can no longer transmit data but the other side can still receive data. (Cheung 2016)

Handling disconnection is much trickier compared to connection. There are multiple states that the TCP connection can be in. It may be that the endpoint has closed the connection but the other endpoint does not know about it. The connection can be disconnected from either endpoint. Luckily Akka.IO handles these cases properly and send messages when these things happen if they're detectable.

It may be good to know when a connection is having problems but is not disconnected. There are no reliable ways but there are good enough methods for this. The connection problems can be detected from drop of bandwidth usage. If either side is sending data at regular intervals the bandwidth drop can be detected from slow rate of receiving. data.

4.2 Detecting TCP disconnection

Perhaps the most challenging part of TCP is reliably detecting when a connection is lost unexpectedly. Akka.IO can detect when a connection is lost with the *ConnectioClosed* message, however, it is not 100% reliable. The reason why this is not reliable is because there is no difference between dead connection and alive connection if there is no network traffic. Neither side can know if the connection is alive unless packets are being sent.

This is a deliberate design decision in the TCP protocol. It allows one end point to restart or change to a physically different device without the loss of connection. Also, if

an end point only reads from connection and never writes, neither side cannot detect loss of connection. (Stephen Cleary 2009)

In order to actually make sure connection is alive, something must be sent at regular intervals and check if the packet is received. The standard way to do this is by sending keep alive packets. If the receiver does not get keep alive packets for a while they know that the connection is dead.

TCP protocol also has integrated keep-alive packet sending. However, it is not recommended to use it since all platforms may not support it as it is not required to be implemented by the TCP spec. (Stephen Cleary 2009)

4.3 Testing TCP disconnection

Losing a connection can be tested by disabling network adapter in operating system. However, this is usually not a realistic scenario because the operating system sends a notification that a connection is lost. Only basic tests can be done with this method.

A more realistic test can be done by unplugging the external connection from the switch. (Figure 8) This does not mean the connection between the computer with the application and switch but the connection between the router and the device which the application is trying connect through.

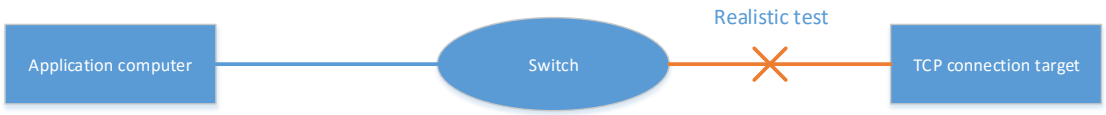

Figure 8 Testing disconnection reliably

4.4 Reading messages

TCP is a streaming protocol, which means that data is sent as a continuous stream. There are no explicit boundaries between messages. Messages can be broken into pieces at any point. The only thing guaranteed is that all pieces are intact and in order.

TCP messages must have a header which tells how long the message is. By knowing the length, the applications can determine when a message starts and ends. The usual process for reading messages goes as follows:

1. Read until minimum amount of data is read to be able to calculate the whole message length
2. Read and store the length
3. Read the length amount of data
4. Store the leftovers for next message

Another method of determining the boundaries is to have some kind of delimiter data. For example, a new line could be used but if this method is used the actual data should not have these delimiters. Having multi-byte delimiters reduces the chance of these kind of errors from occurring. (Karl Seguin 2012)

At every step it is important to take into account that one never knows how much data one is going to get. One may get a stream of single bytes or multiple messages per data block. All cases should be handled correctly.

4.5 Writing messages

Writing messages is a great deal simpler compared to reading. As much as wanted can be written at a time as Akka.IO splits the packets automatically. The writer should also have a header with the message length included or delimiter bytes between messages.

4.6 Interacting with different protocols

Most networked software tends to communicate with some third party. These parties use different protocols which have different requirements. In this section a closer look is taken into few of these protocols and how they should be used with the actor model.

4.6.1 Modbus

Modbus is a communications protocol designed to be used with programmable logic controllers. It is commonly used in industrial electronic devices. In industrial applications Modbus devices are usually controlled through SCADA. Modbus is open and royalty free. (Wikipedia, Modbus 2016)

Modbus operations are done using function codes. Device measurements and statuses lie in registers. Function codes act on those registers. The minimum amount of data that can be read or written from registers is 16 bits. (Wikipedia, Modbus 2016)

There are multiple Modbus protocols available for use. Most of them work through either serial port or Ethernet. Communication with Modbus devices happens through unique address. Devices may act as masters or slaves. Only master devices may act on commands while slave devices replicate masters. (Wikipedia, Modbus 2016)

Many of the devices used in smart traffic field use Modbus as their communication protocol. Modbus libraries exist for many languages so it a good protocol to use.

For C# there is a library for interacting with various Modbus protocols called *NModbus4*. The library was used to build an application to control variable message signs using Akka.NET.

Operations in NModbus4 happen synchronously. Fortunately, the operations are relatively fast so they do not cause problems with Akka.NET as blocking, synchronous operations tend to.

4.6.2 OPC

OPC DA (Open Platform Communications Data Access) is a variable based communication protocol. It is old but well supported. Data in OPC DA is stored in variables. Variables support variety of data types such as integers, strings, floats and multidimensional arrays.

When OPC is referred generically usually OPC DA is meant. OPC DA is the actual variable based read-write standard. OPC also contains various other standards. (Wikipedia, Open Platform Communications 2016)

OPC was designed for communications between Windows based software and industrial systems. SCADA can use OPC to communicate with third party software. (Wikipedia, Open Platform Communications 2016)

OPC Foundation provides .NET implementation of OPC libraries. These libraries were wrapped in Akka.NET actors to take advantage of actor model and to be used in projects. Similar to Modbus, operations in the OPC library are done synchronously and relatively fast.

To effectively use OPC variables in smart traffic solutions a system was developed to automate reading, writing and change tracking of variables. The basic idea is to have classes which represent either both readable and writeable OPC variables, or either of the two. The classes have writeable and readable interfaces depending on their role. It is important that writes and reads only happen through the interfaces because they also include events for notifying when the read or write variables change.

It is trivial to implement the variable classes for single data types but it gets complicated for array types. If there is not some kind of interface through which indexed writes happen the changes can get lost and never get propagated to the OPC server.

The consumers of the OPC variable classes initialize the variables through a factory class which also inserts them to system wide dictionary. Once the consumers have initialized the variables they can write and read to variables through the interfaces.

The system which is responsible for writing writeable variables and reading readable variables to the OPC server subscribes to all variable changes by iterating all variables of the dictionary. This system is powered by the actor model. If a read variable changes in the OPC server, the change is moved to the actual variable instance. The consumer of the specific OPC variable gets event from this change and they can act accordingly.

Same thing happens when write variable changes by the consumer of the variable. The system writes it to the OPC server once the event is triggered.

5 Architecture

While developing software using the actor model several useful patterns emerged or were adopted from existing documentation. The patterns made the codebase more readable.

5.1 Layers

Building abstractions with the actor model is easy. An actor can be placed between service and the consumer, creating an abstraction. The abstracting actor can receive more general and easier to use messages than the implementation. By looking at the actor implementation and the messages it can receive one can get a good idea how the interface is supposed to be used. Building these abstractions is useful as the consumer of service does not need to know the details of how it works.

Following is an example of how this pattern was used in the actual traffic system implementation. (Figure 9)

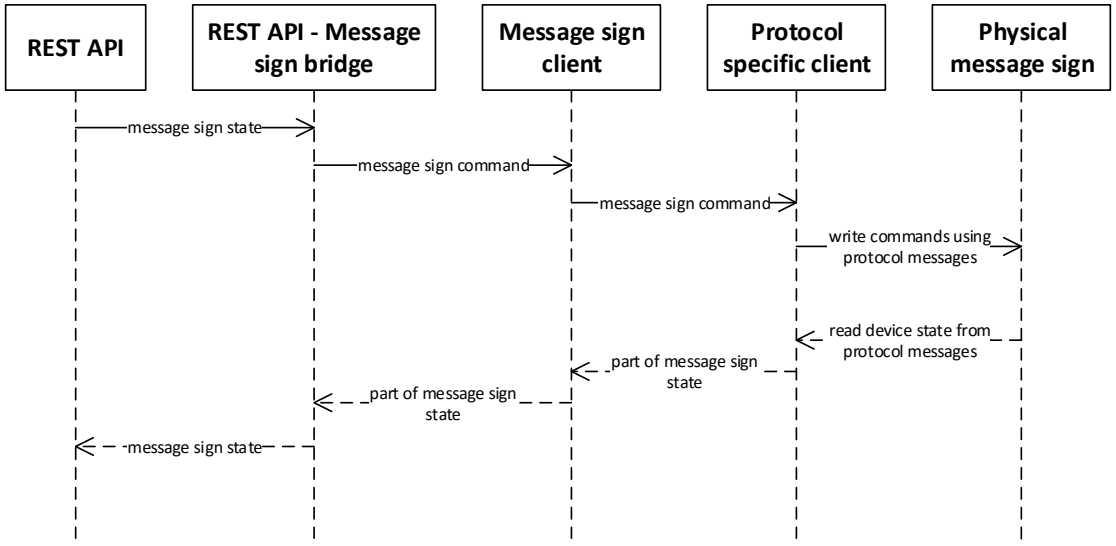

Figure 9 Layers with actor model

The figure shows how controlling a device is handled in terms of layers. The leftmost side is the controlling side, and the rightmost side is the physical device. REST API sends a whole state of the device. The state is then broken into individual messages

and the protocol specific client converts the messages to actual packets that are sent to the physical device.

After a while, the physical device sends its actual state to the protocol specific client. The same process occurs; however, in reverse order, and in the end the device state is formed and sent through REST API.

5.2 Error kernel

As actors are created and a hierarchy is formed a useful pattern which emerges is the error kernel. This pattern is a cornerstone of Akka.NET and is explained in great detail in Akka.NET documentation.

The basic idea is that when an error occurs it is isolated to small area as possible. This is achieved with actor model by splitting work to actors in a way where a similar kinds of failures can only happen in specific child actors. This way when the parent sees the failure, they can decide what to do with the failing actor without affecting other actor's part of the process.

A concrete example of this could be an actor which needs to interact with a database, REST API and a file. This process can be laid out in a way where failure in interaction with one interface does not affect the others. The hierarchy is formed so that each sub process is its own actor and they all have a same supervising parent. (Figure 10)

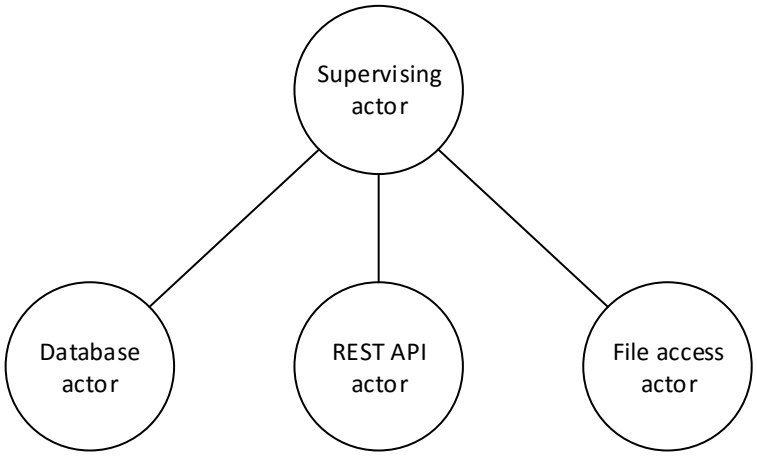

Figure 10 Utilizing error kernel when interacting with interfaces

The supervising actor receives a message from its parent to do some action. The actor then sends messages to every child to do something. If a failure happens in one of the children, the supervisor can either restart the actor, stop it or escalate if the error is unrecoverable. If it is recoverable it does not affect other children the process can continue normally.

5.3 Persistent connection actor

Actor model allows to build a nice abstraction for connection. Libraries which establish a TCP connection can be nicely wrapped in a persistent manner. For example, Modbus and OPC connections can be wrapped in this way. This method also utilizes the error kernel pattern.

The basic idea is to construct an actor which represents the client and the client has a child actor which represents the connection. Users of the service send messages to the client and if the connection happens to be down they can be buffered, which means that from the perspective of the user, the connection is always up. However, it is not always useful to buffer messages as the user of the connection may want to do something special if the connection is down. Example of this method can be seen in figure 11.

```
class Program
{
    static void Main(string[] args)
    {
        var actorSystem = ActorSystem.Create("AkkaIoTest");
        actorSystem.ActorOf<Client>();
        Console.ReadKey();
    }
}

class Client : ReceiveActor
{
    private IActorRef connection;

    public Client()
    {
        Connect();

        Receive<Tcp.Write>(m =>
        {
            // The client could decide here what to
            // do with writes when the connection is not established
            if (connection == null) return false;
            connection.Tell(m);
            return true;
        });

        Receive<Tcp.Connected>(m =>
        {
            // Create the connection actor and watch for lifetime messages
            connection = Context.ActorOf(Props.Create(() => new Connection(Sender)));
            Context.Watch(connection);
            Sender.Tell(new Tcp.Register(connection));
        });

        // If connection fails simply reconnect immediately
        Receive<Tcp.CommandFailed>(
            m => m.Cmd is Tcp.Connect,
            m => Connect());

        // If the connection terminates simply reconnect immediately
        Receive<Terminated>(
            m => Sender == connection,
            m => { connection = null; Connect(); });
    }

    private void Connect() => Context.System.Tcp().Tell(
        new Tcp.Connect(new IPEndPoint(IPAddress.Loopback, 27000)));
}

class Connection : ReceiveActor
{
    public Connection(IActorRef connection)
    {
        Receive<Tcp.Write>(m => connection.Tell(m));
        Receive<Tcp.Received>(m => Context.Parent.Tell(m));
        Receive<Tcp.ConnectionClosed>(m => Context.Stop(Self));
    }
}
```

Figure 11 An example demonstrating Akka.IO and persistent connection actor

5.4 Transactions with actor model

The actor model guaranteeing at-most-once delivery and Akka.NET guaranteeing ordering of messages between two actors in same process lays out a good groundwork for an implementation of transactions.

Start and end of a transaction is denoted with a message. When the start message is received the actor can setup itself in a way which allow the following messages to be rolled back. Once the end message is received the actor can commit all of the received messages so far. This method works even for multiple simultaneous starts and ends because the actor can group transactions by the sender.

A modified version of transaction system was used in controlling of message sign. A start message was used to mark the start of network operation and a timer was started which allows checking of slow connections. In between the start and the end various synchronous Modbus operations were done. If the operations took longer than the timer it was determined that the connection is working slowly and a message notifying of this was sent.

6 Logging

Having good logging is important for debugging software, especially when running in production as usually it is not possible to run proper debuggers in production software and when investigating problems, it usually involves checking why something happened in the past as opposed what is currently happening and how it can be fixed so it doesn't happen again.

It is better to have too much logging than not enough as the logs can be filtered. Log levels should also be utilized because they make filtering easier. The recommended usages for each level is shown on table 1

Table 1. Using error levels

Error level	**Usage**
Error	Errors which may cause software to break
Warning	Errors which never cause software to break but should not happen in production
Info	Information which helps debugging the order of events when investigating errors
Debug	Used mostly for debugging during development

7 Conclusion

The goal was to use the actor model in practice by building networked software utilizing it, develop patterns, figure out the limitations and create guidelines for the company to use. These goals were met for the most part.

The company now has documentation regarding the actor model and architecture used in projects of the company. The actor model was used in practice and limitations were also found. Akka.NET was successfully used in projects and some on them are already in production.

Compared to existing research of the actor model and networking the work doesn't provide significant results to the software development field. Mostly the results consist of practical examples and applications of the existing information about the actor model. However, in smart traffic field the usage of actor model is relatively new and the work can improve the quality of software and give guidelines going forward. The results are usable in other software development fields as they face similar problems.

The results could be improved by comparing other programming languages, frameworks and models to the actor model. There could be patterns and methodologies which could prove to be even more efficient. By using the actor model more, more patterns could also be developed.

Building the projects utilizing the actor model has been fun and I've learned great deal about software development in general and smart traffic. Technologically I've learned more about C#, .NET and various other libraries in addition to the actor model.

References

Akka.NET. 2016. Akka.NET front-page. Accessed on 7 December 2016. Retrieved from http://getakka.net/

Akka.NET. 2016. Serialization. Accessed on 22 May 2016. Retrieved from http://getakka.net/docs/Serialization

Akka.NET. 2016. Why Akka. Accessed on 14 May 2016. Retrieved from http://getakka.net/docs/Why%20Akka

C#. 2016. Wikipedia article. Accessed on 21 December 2015. Retrieved from https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

Carl Hewitt. 2016. Actor Model of Computation: Scalable Robust Information Systems. Accessed on 23 May 2016. Retrieved from https://arxiv.org/ftp/arxiv/papers/1008/1008.1459.pdf

Channel 9. 2016. The Future of C#. Video. Accessed on 23 May 2016. Retrieved from https://channel9.msdn.com/Events/Build/2016/B889

Channel 9. 2012. The Actor Model. Video. Accessed on 14 May 2016. Retrieved from https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

Cheung. 2016. TCP: terminating a connection. Accessed on 23 May 2016. Retrieved from http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/7-transport/tcp3.html

Karl Seguin. 2012. Reading from TCP streams. Accessed on 23 May 2016. Retrieved from http://openmymind.net/2012/1/12/Reading-From-TCP-Streams/

Microsoft Visual Studio. 2016. Wikipedia article. Accessed on 15 May 2016. Retrieved from https://en.wikipedia.org/wiki/Microsoft_Visual_Studio

Modbus. 2016. Wikipedia article. Accessed on 19 May 2016. Retrieved from https://en.wikipedia.org/wiki/Modbus

MSDN. 2016. Introduction to the C# Language and the .NET Framework. Accessed on

15 May 2016. Retrieved from https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx

Open Platform Communications. 2016. Wikipedia article. Accessed on 23 May 2016. Retrieved from https://en.wikipedia.org/wiki/Open_Platform_Communications

Roslyn. 2016. Roslyn overview. Accessed on 11 May 2016. Retrieved from https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview

Soumyasch. 2007. Wikipedia article. Accessed on 7 December 2016. Retrieved from https://commons.wikimedia.org/wiki/File:DotNet.svg

Stephen Cleary. 2009. Detection of Half-Open (Dropped) Connections. Accessed on 12 May 2016. Retrieved from http://blog.stephencleary.com/2009/05/detection-of-half-open-dropped.html

Xavier Decoster. 2013. An Overview of the NuGet Ecosystem. Accessed on 23 May 2016. Retrieved from http://www.codeproject.com/Reference/628210/An-Overview-of-the-NuGet-Ecosystem